دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

مبانی هوش محاسباتی

# پروژه سوم
# بازی تکامل عصبی

هلیاسادات هاشمی پور
۹۸۳۱۱۰۶

تیر ۱۴۰۱

**توضیحات مورد نیاز کد به صورت کامنت واضح بیان شده است**

**هدف از این پروژه استفاده از یک الگوریتم تکاملی برای یادگیری شبکه عصبی در محیطی است که داده های کافی برای یادگیری وجود ندارد.**

## بخش ۱)

در این بخش تابع فعالیت را پیاده سازی کردیم که در دستور کار سیگموید خواسته اما من با توجه به خروجی بهتر، سافت مکس را انتخاب کردم. سپس feedforward را با توجه به وزن و بایاس پیاده‌سازی کردیم.

```
    Neural Network initialization.
    Given layer_sizes as an input, you have to design a Fully Connected Neural Network architecture here.
    :param layer_sizes: A list containing neuron numbers in each layers. For example [3, 10, 2] means that there are
    3 neurons in the input layer, 10 neurons in the hidden layer, and 2 neurons in the output layer.
    """
  # TODO (Implement FCNNs architecture here)
  input_layer, hidden_layer, output_layer = self.layer_sizes  # input_layer = 8, hidden_layer = 32, output_layer
  # = 2
  self.weight_1 = np.random.randn(input_layer * hidden_layer).reshape(hidden_layer,
                                                                      input_layer)  # weight_1 = 32x8
  self.bias_1 = np.zeros((hidden_layer, 1))  # bias_1 = 32x1
  self.weight_2 = np.random.randn(output_layer * hidden_layer).reshape(output_layer,
                                                                       hidden_layer)  # weight_2 = 2x32
  self.bias_2 = np.zeros((output_layer, 1))  # bias_2 = 2x1

ef activation(self, x):
  """
  The activation function of our neural network, e.g., Sigmoid, ReLU.
  :param function:
  :param x: Vector of a layer in our network.
  :return: Vector after applying activation function.
  """
  # TODO (Implement activation function here)

  return np.exp(x) / np.exp(x).sum()
  # return 1 / (1 + np.exp(-x))

ef forward(self, x):
  """
  Receives input vector as a parameter and calculates the output vector based on weights and biases.
  :param x: Input vector which is a numpy array.
  :return: Output vector
  """
  # TODO (Implement forward function here)
  x = np.array(x)
  x = x.reshape(self.layer_sizes[0], 1)  # x = 8x1
  z1 = self.weight_1 @ x + self.bias_1  # z1 = 32x1
  a1 = self.activation(z1)  # a1 = 32x1
  z2 = self.weight_2 @ a1 + self.bias_2  # z2 = 2x1
  a2 = self.activation(z2)  # a2 = 2x1
  return z2  # return 2x1 (z2)
```

## بخش ۲)

در این بخش با توجه به حدس و خطایی که داشتیم و یادگیری که باید باشد (با توجه به تابع فعالیت) خط ۳۸ را ست کردیم تا پاسخ به مراتب بهینه شود.(که من ۸ تا نورون لایه ورودی، ۳۲ تا نورون برای لایه پنهان و ۲ تا هم برای لایه خروجی در نظر گرفتم). برای تابع think را هم با توجه به ورودی‌هایی که دارد ابتدا باید بر روی پرش بازیکن ها کار کنیم، یعنی با توجه به زمین بازی و موانع موجود پرش به چپ یا راست بازیکن را پیاده‌سازی کردیم. پس از تشکیل بردار ورودی هم با توجه به تابع پیاده‌سازی شده برای feedforward خروجی شبکه عصبی را تولید کرده و با توجه به آن change_garvity را فراخوانی می کنیم.

```
        layer_sizes = [8, 32, 2]  # TODO (Design your architecture here by changing the values)
        self.nn = NeuralNetwork(layer_sizes)
```

```
def batch_func(self,coordination):
    coordination = np.array(coordination)  # to change the input vector of the neural network (coordination)
    var = sum((coordination - sum(coordination) / len(coordination)) ** 2) / len(
        coordination)  # to change the input vector of the neural network (var)
    coordination = (coordination - sum(coordination) / len(coordination)) / (
        math.sqrt(var + 1e-5))  # to change the input vector of the neural network (coordination)
    return coordination

def batch_learning_nn(self, screen_height, obstacles, player_x, player_y):
    coordination = [player_x,
                    player_y]  # change the value of coordination here to change the input vector of the neural
    # network (x, y)
    for i in range(
            3):  # change the value of i here to change the number of obstacles in the input vector (3 is the
        # default)
        try:  # IndexError: list index out of range
            y = obstacles[i]['y']  # change the value of y here to change the input vector of the neural network (y)
            # print(y)
            x = obstacles[i]['x']  # to change the input vector of the neural network (x)
            # print(x)
            # print(player_y) #640
            # print(player_x)
            if y - 2 < player_y:  # if obstacle is above player
                coordination.append((screen_height - y))  # to change the input vector of the neural network (y)
                coordination.append(x)  # to change the input vector of the neural network (x)
            else:  # if obstacle is below player
                coordination.append(1)  # to change the input vector of the neural network (y)
                coordination.append(1)  # to change the input vector of the neural network (x)
        except:  # IndexError: list index out of range # if there are less than 3 obstacles
            coordination.append(1)
            coordination.append(1)

    return self.batch_func(coordination)  # (change the value of coordination here)
```

```
    input = self.batch_learning_nn(screen_height, obstacles, player_x, player_y)
    try:
        output = self.nn.forward(input)
        if output[0] > 1 / 2:  # if output[0] is greater than 0.5 then the player should jump
            # print(f"Gravity changed to left for {self.player_index}")
            self.change_gravity('left')  # change gravity to left
        elif input[1] > 1 / 2:  # if input[1] is greater than 0.5
            self.change_gravity('right')  # change gravity to right
            # print(f"Gravity changed to right for {self.player_index}")
    except:
        pass
```

## بخش۳)

در این بخش در ورودی تابع next_population_selection یک لیستی از بازکنان گرفته به همراه تعداد آن ها، حال ابتدا لیست بازکنان را سورت کرده (باتوجه به مقدار شایستگی) سپس max،minو میانگین را محاسبه می کنیم به عنوان خروجی هم با توجه به عملکردی که هر کدام از qtornoment،sus و یا چرخ رولت (که هر کدام را با تجوه به عملکرد خاص خودشان پیاده سازی کردم)دارند باید بازماندگان را به عنوان خروجی برگردانیم.که در اینجا من sus را برای فرزندان انتخاب کردم. در اصل با ارزیابی هر کدام از توابع موجود با sus خروجی بهتری داشتیم

```python
def next_population_selection(self, players, num_players):
    """
    Gets list of previous and current players (μ + λ) and returns num_players number of players based on their
    fitness value.

    :param players: list of players in the previous generation
    :param num_players: number of players that we return
    """
    # TODO (Implement top-k algorithm here)
    # TODO (Additional: Implement roulette wheel here)
    # TODO (Additional: Implement SUS here)
    # TODO (Additional: Learning curve)
    s_players = sorted(players, key=lambda player: player.fitness, reverse=True)  # Sort by fitness (highest to
    # lowest)
    max = s_players[0].fitness  # max(players)
    min = s_players[len(s_players) - 1].fitness  # min(players)
    average = sum([player.fitness for player in players]) / len(
        [player.fitness for player in players])  # average fitness of the population
    print([min, max, average])  # Print the min, max, and average fitness of the population
    self.accuracy.append((min, max, average))  # Add the min, max, and average fitness to the accuracy list (for
    # plotting)
    self.mutate_num = 0  # Reset the number of mutations
    # data = pd.DataFrame(np.array(evolution.accuracy), columns=["min", "max", "avg"])
    # csv_name = "generation_analysis_" + datetime.now().strftime("%d%H%M%S") + ".csv"
    # data.to_csv(csv_name)

    return self.roulette_wheel(players, num_players)  # Return the next population based on the selection method
```

```python
def roulette_wheel(self, players, parent_numbers):  # Roulette wheel selection
    probabilities = self.cal_cumulative_probabilities(players)  # Calculate cumulative probabilities
    new_population = []  # List of players that will be returned
    for temp in np.random.uniform(low=0, high=1, size=parent_numbers):  # parent_numbers is the number of parents
        # that we want to return
        for i, probability in enumerate(probabilities):  # Find the player that corresponds to the probability  (
            # i.e. the parent)
            if temp <= probability:  # If the probability is greater than the probability of the player
                res = self.clone_player(players[i])  # Clone the player and add it to the new population
                new_population.append(res)  # Add the player to the new population and break the loop
                break  # Break the loop (We only need to find the first player that corresponds to the probability)
    return new_population  # Return the list of players
```

```python
def sus(self, players, num_players):  # SUS selection
    probabilities = self.cal_cumulative_probabilities(players)  # Calculate cumulative probabilities
    new_population = []  # List of players that will be returned
    move = (probabilities[len(probabilities) - 1] - np.random.uniform(0, 1 / num_players,
                                                                      1)) / num_players  # Step size for SUS
    # selection (1/num_players) is the step size for the roulette wheel selection (0.1)
    for i in range(num_players):  # num_players = num_players // q_size
        temp = (i + 1) * move  # Calculate the probability of the next player to be selected
        for j, probability in enumerate(probabilities):  # Find the player that corresponds to the probability
            if temp <= probability:  # If the probability is greater than the probability of the player
                res = self.clone_player(players[i])  # Clone the player and add it to the new population
                new_population.append(res)  # Add the player to the new population and break the loop
                break  # Break the loop (We only need to find the first player that corresponds to the probability)
    return new_population  # Return the new population of players
```

```python
def q_tournament(self, players, num_players, q_size=2):  # Q-tournament selection
    next_population = []  # List of players that will be returned (μ + λ)
    for i in range(num_players):  # num_players = num_players // q_size
        temp_population = []  # list of players that will be used to calculate fitness values for the next
        # generation
        for j in range(q_size):  # q_size is the size of the tournament (2 by default)
            temp_population.append(players[np.random.randint(0, len(players))])  # random player from the population
        temp_population.sort(key=lambda x: x.fitness, reverse=True)  # Sort by fitness (highest to lowest)
        next_population.append(temp_population[0])  # Take the best player and add it to the next population
    return next_population  # Return the next population based on the previous population
```

## بخش۴)

در این بخش باید والدین را انتخاب کنیم. ابتدا تابع crossover و mutation را پیاده سازی می‌کنیم. به این صورت عمل کرده که ابتدا تمامی بازماندگان را والد در نظر گرفته و حالت هایی که داریم را با توجه به اینکه هر کدام عملکرد بهتری دارند را در نظر می گیریم سپس دو به دو والدین را انتخاب کرده و فرزندان را هم با توجه به توابع پیاده سازی شده برای جهش و تقاطع تولید می کنیم. در آخر هم به عنوان خروجی لیست فرزندان را برگرداندیم. که چرخ رولت برای والدین با توجه به خروجی که می داد، انتخاب کردم.

```
def mutation(self, child, threshold):  # Mutation function (Mutation rate = threshold)
    chance = np.random.uniform(0, 1, 1)  # Random chance to mutate (0-1)
    if chance < threshold:  # If the chance is less than the threshold (mutation probability)
        self.mutate_num += 1  # Increment the number of mutations by 1 (for the next generation)
        child.nn.weight_1 += np.random.randn(child.nn.weight_1.shape[0] *
                                             child.nn.weight_1.shape[1]).reshape(child.nn.weight_1.shape[0],
                                                                                child.nn.weight_1.shape[1])  #
        # Add random noise to the weights (gaussian distribution)
    chance = np.random.uniform(0, 1, 1)
    if chance < threshold:
        self.mutate_num += 1
        child.nn.weight_2 += np.random.randn(child.nn.weight_2.shape[0] *
                                             child.nn.weight_2.shape[1]).reshape(child.nn.weight_2.shape[0],
                                                                                child.nn.weight_2.shape[1])
    chance = np.random.uniform(0, 1, 1)
    if chance < threshold:
        self.mutate_num += 1
        child.nn.bias_1 += np.random.randn(child.nn.bias_1.shape[0] *
                                           child.nn.bias_1.shape[1]).reshape(child.nn.bias_1.shape[0],
                                                                             child.nn.bias_1.shape[1])
    chance = np.random.uniform(0, 1, 1)
    if chance < threshold:
        self.mutate_num += 1
        child.nn.bias_2 += np.random.randn(child.nn.bias_2.shape[0] *
                                           child.nn.bias_2.shape[1]).reshape(child.nn.bias_2.shape[0],
                                                                             child.nn.bias_2.shape[1])
```

```
def operations(self, parent1, parent2):  # Operations
    threshold = 0.3  # Threshold for the mutation probability
    child1 = self.clone_player(parent1)  # Clone the first parent
    child2 = self.clone_player(parent2)  # Clone the second parent
    # weights
    self.crossover(child1.nn.weight_1, child2.nn.weight_1, parent1.nn.weight_1, parent2.nn.weight_1)  # Crossover
    self.crossover(child1.nn.weight_2, child2.nn.weight_2, parent1.nn.weight_2, parent2.nn.weight_2)  # Crossover
    # biases
    self.crossover(child1.nn.bias_1, child2.nn.bias_1, parent1.nn.bias_1, parent2.nn.bias_1)  # Crossover
    self.crossover(child1.nn.bias_2, child2.nn.bias_2, parent1.nn.bias_2, parent2.nn.bias_2)# Crossover
    # mutation
    self.mutation(child1, threshold)
    self.mutation(child2, threshold)
    return [child1, child2]
```

```python
def crossover(self, child1, child2, parent1, parent2):  # Crossover (uniform crossover)
    section_1 = int(child1.shape[0] / 3)  # Section 1" is the first third of the child
    section_2 = int(2 * child1.shape[0] / 3)  # Section 2" is the second third of the child
    rnd = np.random.uniform(0, 1, 1)  # Random number between 0 and 1

    if rnd < 0.5:  # If the random number is less than 0.5
        child1[:section_1, :] = parent2[:section_1:,
                                :]  # Copy the first third of the parent2 to the first third of the child1
        child1[section_1:section_2, :] = parent1[section_1:section_2,
                                         :]  # Copy the second third of the parent1 to the second third of the child1
        child1[section_2:, :] = parent2[section_2:,
                                :]  # Copy the third third of the parent2 to the third third of the  child1
        child2[:section_1, :] = parent1[:section_1:,
                                :]  # Copy the first third of the parent1 to the first third of the child2
        child2[section_1:section_2, :] = parent2[section_1:section_2,
                                         :]  # Copy the second third of the parent2 to the second third of the child2
        child2[section_2:, :] = parent1[section_2:,
                                :]  # Copy the third third of the parent1 to the third third of the child2
    else:
        child1[:section_1, :] = parent1[:section_1:,
                                :]  # Copy the first third of the parent1 to the first third of the child1
        child1[section_1:section_2, :] = parent2[section_1:section_2,
                                         :]  # Copy the second third of the parent2 to the second third of the child1
        child1[section_2:, :] = parent1[section_2:,
                                :]  # Copy the third third of the parent1 to the third third of the child1
        child2[:section_1, :] = parent2[:section_1:,
                                :]  # Copy the first third of the parent2 to the first third of the child2
        child2[section_1:section_2, :] = parent1[section_1:section_2,
                                         :]  # Copy the second third of the parent1 to the second third of the child2
        child2[section_2:, :] = parent2[section_2:,
                                :]  # Copy the third third of the parent2 to the third third of the child2
```

```python
def generate_new_population(self, num_players, prev_players=None):
    """
    Gets survivors and returns a list containing num_players number of children.

    :param num_players: Length of returning list
    :param prev_players: List of survivors
    :return: A list of children
    """
    first_generation = prev_players is None  # If the previous generation is None
    if first_generation:  # If this is the first generation
        return [Player(self.game_mode) for _ in
                range(num_players)]  # Return a list of num_players number of children
    else:  # If this is not the first generation
        prev_parents = prev_players.copy()  # Copy the previous generation
        prev_parents = self.sus(prev_parents, len(prev_parents))  # Get the survivors
        children = []  # Initialize the children list
        for i in range(0, len(prev_parents), 2):  # For each pair of parents
            children += self.operations(prev_parents[i], prev_parents[i + 1])  # Get the children
        return children  # Return the children
```

## بخش۵)

به عنوان بهش امتیازی هم یک فایل در نظر گرفته و مشخصات خواسته شده اعم از بیشترین، کم ترین و میانگین شایستگی بازیکنان را در آن ذخیره کردیم و در آخر هم پلات کردیم.

```
data = pd.DataFrame(np.array(evolution.accuracy), columns= ["min", "max", "avg"])
csv_name = "generation_analysis_" + datetime.now().strftime("%d%H%M%S") + ".csv"  # Create a name for
# the csv file using the current time and date
data.to_csv(csv_name) # Save the dataframe to a csv file
```